1987

Execution of Real-Time Prototypes

Luqi

Naval Postgraduate School

Luqi, "Execution of Real-Time Prototypes", Technical Report NPS 52-87-012,
Computer Science Department, Naval Postgraduate School, 1987.
http://hdl.handle.net/10945/65240

T 10

NPS52-87-012

NAVAL POSTGRADUATE SCHOOL

Monterey, California

EXECUTION OF REAL-TIME PROTOTYPES

LuQi

April 1987

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

D. A. Schrady
Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

[redacted]
LUQI
Assistant Professor
of Computer Science

Reviewed by:

[redacted]
VINCENT Y. LUM
Chairman
Department of Computer Science

Released by:

[redacted]
KNEALE T. MARSHALL
Dean of Information and
Policy Science

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-87-012	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): EXECUTION OF REAL-TIME PROTOTYPES	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): LuQi	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61153N: RR014-01, N0001487 WR4E011
11. CONTROLLING OFFICE NAME AND ADDRESS: Chief of Naval Research, Arlington, VA 22217	12. REPORT DATE: April 1987
	13. NUMBER OF PAGES: 16
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

PSDL is a language for rapid prototyping, especially for systems with real-time constraints. This paper briefly introduces prototyping and PSDL, and describes how a PSDL prototype can be used to firm up the requirements of a software system. A scheme for making PSDL prototypes executable is outlined and the required support software is described.

Execution of Real-Time Prototypes

Luqi

Computer Science Department

Naval Postgraduate School

Monterey, CA 93943

ABSTRACT

PSDL is a language for rapid prototyping, especially for systems with real-time constraints. This paper briefly introduces prototyping and PSDL, and describes how a PSDL prototype can be used to firm up the requirements of a software system. A scheme for making PSDL prototypes executable is outlined and the required support software is described.

1. Introduction

Some important problems facing the computer software industry are achieving cost effective production of software systems and increasing the quality of software products with respect to meeting user requirements. Many software development methodologies have been proposed to approach this goal. Most of the well-known ones, such as Object Oriented Design (OOD) and the Jackson System Development Method (JSD) depend on the skill of individual designers at the level of manual work. These methods are labor intensive, and are too informal to guarantee any quality standards for the resulting design. These methods are unlikely to lead to any significant improvements in the reliability of software products.

Newly proposed software tools for software design, such as GENESIS [Ramamoorthy 85], SREM [Alford 77], the PAISLey operational approach [Zave 82], and DIANA [Evans 83] go one step further. Most of them are really software development environments consisting of many software tools for computer-aided software development. Some of them are designed to fit specific needs, eg. DIANA is a tool for the design of Ada systems. Even if we consider only the prototyping aspects of these tools, it is clear that complete automation of software development is still a distant goal. Most of these types of tools cannot be extended to the point of complete automation because of the lack of mathematical formalization in the related theoretical fields of software engineering.

In the rapid prototyping paradigm, the traditional software life cycle used in software design is replaced by a recently proposed alternative life cycle which consists of two phases: rapid prototyping and automatic program generation [Yeh 84]. Completely automatic generation of programs from very high level specifications is not currently practical. In our approach, program construction is speeded up by taking advantage of reusable software components drawn from a software base. The aspects of program construction that benefit from mechanical assistance are retrievals from the software base, generation of code for interconnecting available modules, and static task scheduling.

A rapid prototype is an executable model or a pilot version of the intended system. The construction activity leading to such a rapid prototype is called rapid prototyping [Neumann 82]. The use of rapid prototyping in requirements analysis

is described in Figure 1 [Luqi 87d].

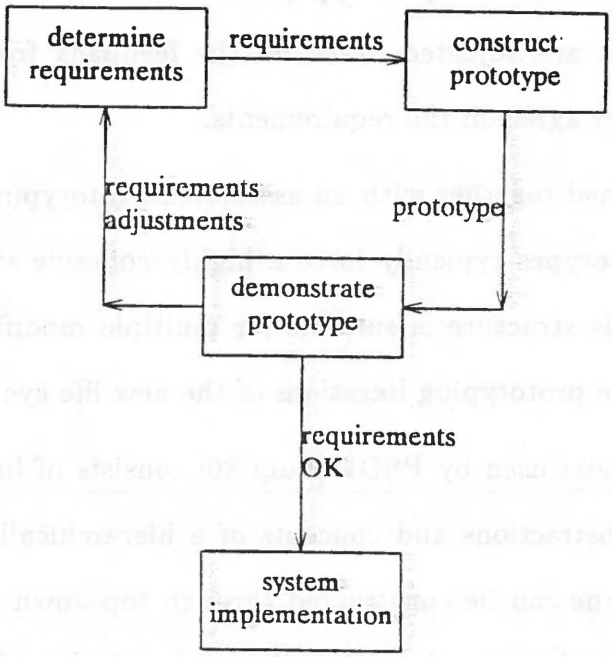

Figure 1 - Process of requirments determination and validation by prototyping

PSDL (Prototype System Description Language) has been designed for prototyping large software systems at the University of Minnesota. The language has special features particularly appropriate for real time system design.

PSDL prototypes can be used for requirements analysis and validation. A key issue in the design of large software systems is how to meet the requirements. Lack of agreement on the requirements as specified by user and analyzed by the designer usually increases the development costs and causes inconsistencies between the delivered system and user expectations [Yeh 83]. Because the user can usually recognize whether or not a working software system does what is needed, but usually can't describe the requirements accurately, prototypes are an effective means for achieving stable and accurate requirements. PSDL prototypes are especially well suited for this purpose because they are executable and are constructed at a specification level with requirements tracing. The prototype is used

in an iterative process of negotiation. The user describes the requirements, the designer interprets them and builds a prototype, and then asks user to criticize the result. The requirement are adjusted based on the feedback from the users until both users and designer agree on the requirements.

Since PSDL was designed together with an associated prototyping methodology [Luqi 87b], PSDL prototypes typically have a highly cohesive structure and few coupling problems. This structure is suitable for multiple modifications at a specification level during the prototyping iterations of the new life cycle.

The computational model used by PSDL [Luqi 86] consists of basic building blocks for describing the abstractions and concepts of a hierarchically structured prototype. A PSDL prototype can be constructed through top-down design. The PSDL non-procedural control constraints describe and enhance the real-time requirements at all layers of the hierarchical prototype.

The executable PSDL prototype is also used to check real-time requirements because the critical timing constraints and the most important concerns, eg. maximum execution time, minimum response time, and synchronization are very hard to validate without actually constructing a valid schedule and observing the execution of the prototype. Most real time systems are used to monitor and control physical processes external to the computer in the embedded system. The precision and accuracy requirements in the design of a real time control system complicate the demands on the execution of the designed software system. For these reasons, the design of real time systems imposes a different set of demands. The formal structure specifying the real time constraints in PSDL provides a basis for automating the production of code from the formal requirements specifications to the underlying programming language. The execution of PSDL prototypes helps to verify that the design of an embedded system with given timing constraints for

the components in the prototype will interact with its environment in a way that meets the timing constraints of the system as a whole. This is important because making a production quality implementation is very expensive, so that it is necessary to check that a design is feasible by using an inexpensive prototype before committing significant resources to an implementation.

PSDL is a language for describing prototypes of large software systems with real-time constraints [Luqi 86]. Such systems are modeled in PSDL as networks of operators communicating via data streams. The data streams can carry data values of an abstract data type [Guttag 78] as well as tokens representing exception conditions. Each type or operator is either composite or atomic. Composite operators are implemented by decomposing them into networks of more primitive operators using PSDL. Atomic operators are realized by retrieving an implementation from a software base [Yeh 84, Roussopoulos 85] containing reusable software components implemented in an underlying programming language. The reusable software components in the software base can be written in any general purpose programming language, provided that PSDL specifications for each module are included.

2. Prototype Execution

PSDL prototypes are executable if all required information is supplied [Luqi 87d] as indicated in Figure 2, CAPS (Computer Aided Prototyping System) Architecture, and the software base contains both normalized specifications and

implementations for all atomic operators and types [Luqi 87c].

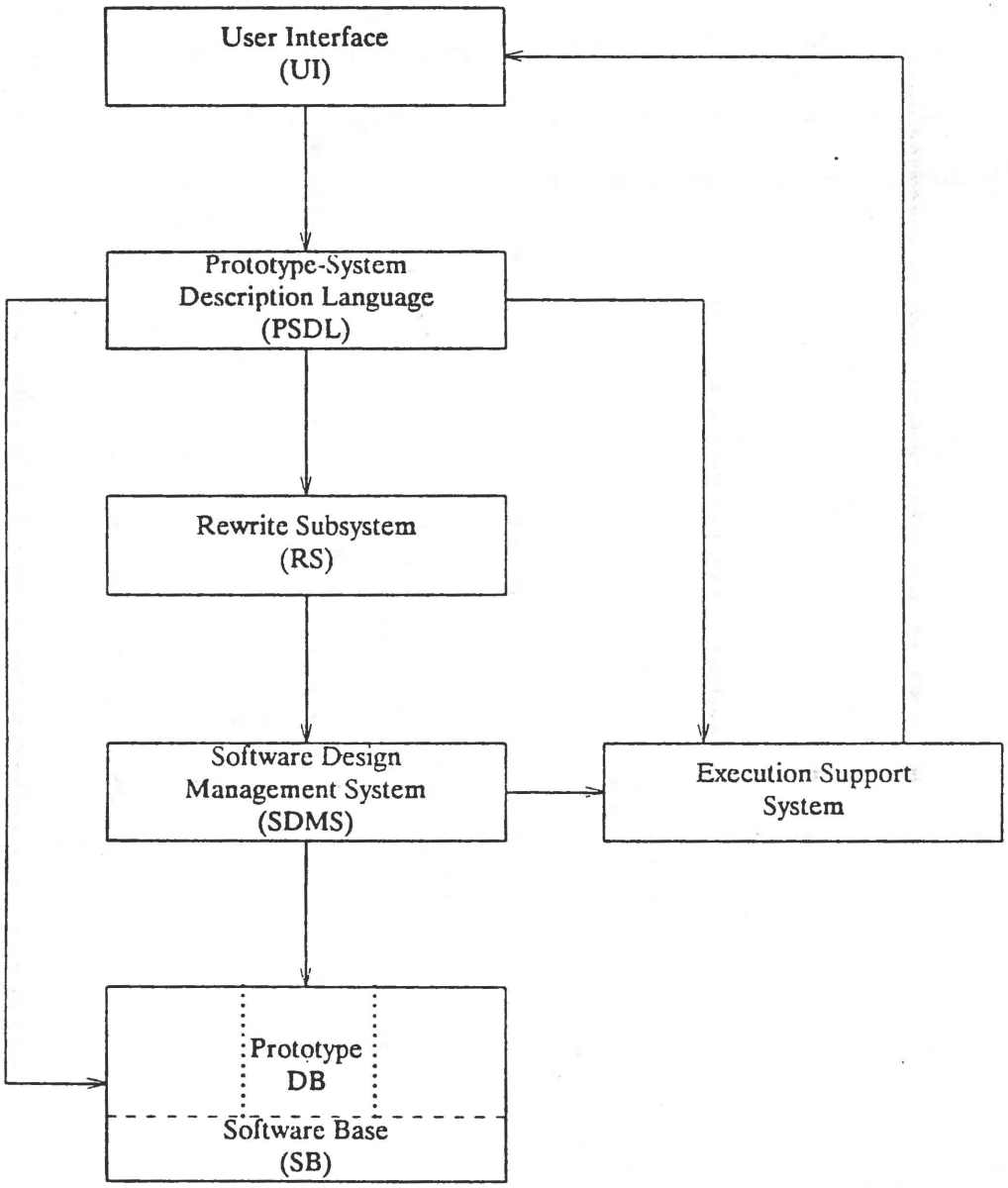

Figure 2- CAPS Architecture

To simplify the description of the PSDL translator we will assume here that Ada [Ada manual] is used for implementing both the reusable components in the software base and the PSDL execution support environment. The PSDL execution support system contains a static scheduler, a translator, and a dynamic scheduler. The static scheduler attempts to find a static schedule for the operators

with real time constraints. The translator augments the implementations of the atomic operators and types with code realizing the data streams and activation conditions, resulting in a program in the underlying programming language that can be compiled and executed. Execution is under the control of a dynamic scheduler, which also provides facilities for debugging and gathering statistics.

2.1. Static Scheduler

The static scheduler analyzes the real time constraints declared in the PSDL prototype, and attempts to find a static schedule meeting the timing constraints of the time critical operators. The operators that do not have real time constraints are handled by the dynamic scheduler, as explained in the next section. Time critical operators are either periodic or sporadic. Sporadic operators are implemented by their periodic equivalents.

The static scheduler partitions the set of periodic operators into non-overlapping HARMONIC BLOCKS, and constructs a STATIC SCHEDULE for each harmonic block. A harmonic block is a set of operators with the following properties:

(1) The periods of all of the operators in the set are exact multiples of the BASE PERIOD.

(2) One of the operators in the set has a period equal to the base period.

Harmonic blocks are not automatically disjoint (contrary to the assumption in [Mok 84a]). We have chosen to make them disjoint by placing each operator that satisfies the constraints for more than one harmonic block into the block with the longest possible base period, since we believe this heuristic will help to ease schedule congestion. A static schedule is a table giving the starting times and execution times for each operator in a harmonic block, and covers a length of time

equal to the least common multiple of all the periods in the block. The static schedule is constructed using algorithms similar to those of [Mok 84a]. The static scheduler uses a set of assumptions about the number and speed of the processors available and the cost of interprocessor communication, which must be provided by the designer.

Note that we treat each harmonic block as an independent scheduling problem. This is justified by the assumption that there will be at least one physical processor for each harmonic block. This assumption is reasonable because any schedule containing two operators with relatively prime periods is guaranteed to have periodic tight spots due to the beats between the two frequencies, leading to low utilization of the scheduled processor.

Sporadic operators are implemented as their periodic equivalents [Mok 84b]. The period P of such an equivalent periodic operator is given by the formula

$$P = \text{MIN}(\text{mcp}, \text{mrt} - \text{met})$$

where mcp is the MINIMUM CALLING PERIOD, mrt is the MAXIMUM RESPONSE TIME, and met is the MAXIMUM EXECUTION TIME. An equivalent periodic operator derived in this way has a deadline equal to the maximum execution time, which means that it must be scheduled to start at the beginning of each period (although a constant phase shift is allowable). A sporadic operator cannot meet its timing constraints unless P is greater than the maximum execution time of the operator.

2.2. Translator

The translator augments the code for the atomic operators to adapt them to the context in which they are going to operate. The augmentations have four main purposes: to implement PSDL data streams, to implement PSDL

conditionals, to implement PSDL timers, and to create exception closures.

Data streams are implemented using buffers. The buffers are at the receiving end of each data stream, and have a capacity of one data item each. The buffer for a sampled stream has a flag which is set whenever a new data value is received and is cleared whenever the data value is read. This flag is used in controlling data driven operators. The buffer for a data flow stream has a similar flag, which is set when the queue is full and cleared when it is empty. The difference between the two kinds of buffers is that an exception occurs if an attempt is made to write into a full data flow buffer or to read from an empty one, while reading or writing a sampled buffer will never cause an exception. The exceptions produced by the incorrect use of data flow buffers cause a transfer of control to the PSDL debugger.

PSDL conditionals are implemented using the conditionals of the underlying programming language. Triggering conditions are implemented by code of the following form:

IF triggering_condition THEN operator_body END

This is followed by an epilog which clears the input buffers and writes the outputs of the operator into the appropriate output buffers. If PSDL output guards are present, the output buffer operations are embedded in conditionals of the form

IF output_guard THEN output_buffer_operation END

The buffers are instances of a built-in data type which is implemented using a standard reusable module written in the underlying programming language. The multiple processor version of the PSDL translator uses a buffer type whose primitive operations are embedded in a monitor to provide mutual exclusion. The buffers are the only source of potential interference between processes, since all of the other data structures are local to a single process. The single processor

implementation does not need monitors for the buffers, because a simple implementation scheme using a single thread of control is most efficient for this case.

PSDL timers are implemented by a standard library package that communicates with a hardware clock. This package is included in any prototype that uses timers.

An exception closure is a package that handles all of the exceptions that may occur inside it. The PSDL translator produces exception closures by augmenting the code for atomic operators with handlers for any exceptions they may raise. These default handlers catch the exceptions and use the CREATE operation of the built-in EXCEPTION data type to create the required exception value, which is transmitted along the appropriate output data streams.

2.3. Dynamic Scheduler

The dynamic scheduler is a run-time executive with three main purposes: to schedule operators that are not time critical, to provide debugging facilities, and to gather statistics about the run-time characteristics of the prototype. In the case of a distributed implementation, there is an instance of the dynamic scheduler running on each processor.

PSDL assumes that time constraints are absolute if they are given. This requires the static scheduler to allocate processor time based on worst case execution times and firing frequencies. This policy results in plenty of spare processor time on the average, because worst case loads tend to be rare. The dynamic scheduler uses a simple strategy to utilize this spare capacity for operations that are not time critical.

During each base period the dynamic scheduler invokes the time critical operators in the order in which they are scheduled. When it runs out of things to

do, it checks to see if it has any time left, and if so it picks a non time-critical operator to execute. A simple round robin scheduling algorithm is used. Just before the end of the base period, the currently running operator is interrupted and the resumption point for the operator is saved. The interrupt is given sufficiently long before the end of the base period so that the currently running operator will have enough time to get out of any critical sections it may have entered. The only critical sections in the system are in the buffering primitives for reading values from data streams and writing values into data streams. These critical sections are short, and have fixed upper bounds on their execution times.

The debugging facilities are fairly conventional. Breakpoints can be attached to operators, and can be conditional with respect to a PSDL predicate. Selected inputs or outputs of an operator can be traced, resulting in a display of the values and their associated arrival or departure times. Commands for inserting and deleting values in data streams are provided. The facilities for gathering statistics include commands for monitoring both frequencies and timing information. Frequency statistics include the number of values that pass down a data stream, the number of times an operation fires, the number of times an exception occurs, etc. Timing statistics include minimum, average, standard deviation, and maximum times for the execution, response, or interval between firing of an operator. These statistics are intended primarily for feasibility and performance studies.

3. Conclusions

There are many aspects of software requirements which can be most effectively validated by user inspection of a running prototype, such as the appropriateness of a given user interface, or the correct description of an existing hardware interface. Executing prototypes of the novel or difficult parts of a com-

plicated system can significantly increase the confidence that the system can in fact be built, before significant resources have been committed to the development effort. Cost estimates can be improved by using a prototype, since the cost of designing the intended system is usually proportional to the cost of the rapid prototype. Performance bottlenecks can be found during the execution of the prototype by collecting statistics on module execution frequencies.

Our initial investigation leads us to conclude that an execution support system for PSDL is feasible, and that such a software tool is currently the most practical way to support rapid prototyping for real-time systems. This together with the features of PSDL for large scale software design makes PSDL a good candidate for inclusion in an advanced Ada programming environment. At the current point in time, we have a conceptual design for the PSDL execution support system, and the implementation of the PSDL translator is under way.

The PSDL language, its associated methodology, and programming environment apply well to the design of Ada software systems. The demand for large scale Ada software systems is increasing dramatically. Real time systems have particularly strict requirements on accuracy and precision. A rapid prototyping environment for creating and modifying an executable prototype is needed. The design of PSDL, its prototyping methodology, and the use of reusable components from a software base make highly automated software tools practical. An experienced PSDL user should be able to rapidly construct a prototype significantly faster than an experienced Ada-user.

The use of PSDL for prototype construction should be much easier and simpler than the direct use of Ada. PSDL has selected and transformed all the good language features of Ada primitive constructs into a small and a simple set of PSDL language constructs which is convenient for the designer. It is simpler to

describe the structure of a system and the relation between system components in PSDL than in Ada since PSDL allows a designer express his thoughts at a specification or a design level. The abstractions of PSDL are tailored to describing real-time systems, and allow the designer to express his thoughts clearly and quickly by eliminating many lower level details from his consideration. The computational model of PSDL forces all interactions between models to be explicit. All state variables are local to some component, thus confining the effects of state changes. This helps designer understanding by eliminating hidden interactions on the large scale, while allowing the efficiencies of imperative programming inside individual components. The important points are that the software tools and the prototyping methodology of PSDL lead to a well structured prototype and that the resulting PSDL prototype is executable. PSDL components can be mapped into Ada directly. Ada is a large and powerful programming language. It is a good underlying programming or an implementation language for PSDL. However, it is too hard and too cumbersome to use as a design language. The mapping between PSDL and Ada and the use of the reusable Ada components are the keys to making PSDL prototypes executable and useful in large Ada projects.

4. References

[Ada manual]

Ada Programming Language, DoD, ANSI/MIL-STD-1815A, 1983.

[Alford 77]

A. W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements", IEEE Transactions on Software Engineering, vol. SE-3, no. 1, 60-68, Jan 1977.

[Evans 83]

A. Evans, K. J. Butler, G. Goos, and W. A. Wulf, "DIANA Reference Manual", Revision 3,

Tartan Laboratories Inc., Pittsburgh, Penn., 1983.

[Guttag 78]

J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract Data Types and Software Validation", CACM 21, 12, Dec. 1978.

[Luqi 86]

Luqi "Rapid Prototyping for Large Software Systems", Ph. D. Thesis, University of Minnesota, May 1986.

[Luqi 87a]

Luqi, V. Berzins, R. Yeh, "A Prototyping Language for Real-Time Software", to appear in IEEE TSE, 1987.

[Luqi 87b]

Luqi, V. Berzins, "Rapid Prototyping of Real-Time Software", revised for IEEE SOFTWARE, 1987.

[Luqi 87c]

Luqi "Normalized Specifications for Identifying Reusable Software", submitted to ACM-IEEE Computer Society 1987 Fall Joint Computer Conference, October 1987.

[Luqi 87d]

Luqi, M. Ketabchi, "A Computer Aided Prototyping System", submitted to First International Workshop on Computer-Aided Software Engineering, May, 1987.

[Mok 84a]

A. K. Mok, "The Design of Real-Time Programming Systems Based on Process Models", IEEE, 1984.

[Mok 84b]

A. K. Mok, "The Decomposition of Real-Time System Requirements into Process Models", IEEE Proc. of the 1984 Real Time Systems Symposium, Dec. 1984, pp. 125-133.

[Neumann 82]

P. G. Neumann, ed., "Special Issue on Rapid Prototyping", SIGSOFT, Dec 82.

[Ramamoorthy 85]

C. V. Ramamoorthy, Y. Usuda, W. Tsai, and A. Prakash, "GENESIS: An Integrated Environment for Supporting Development and Evolution of Software", PROC. COMPSAC 85.

[Roussopoulos 85]

N. Roussopoulos, "Architectural Design of the SBMS", Quarterly Report for the STARS SB/SBMS Project, Dept. of Computer Science, University of Maryland, April 1985.

[Yeh 83]

R. T. Yeh, "Software Engineering", IEEE Spectrum, Nov. 1983, p. 91-94.

[Yeh 84]

R. T. Yeh, R. Mittermeir, N. Roussopoulos, and J. Reed, "A Programming Environment Framework Based on Reusability", Proc. Int. Conf. on Data Engineering, Apr. 1984.

[Zave 82]

P. Zave, "An Operational Approach to Requirements Specifications for Embedded Systems", IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, 250-269, 1982.

Initial Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analyses 2000 N. Beauregard Street Alexandria, VA 22311	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
LuQi Code 52Lq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	100
Office of Naval Research 800 N. Quincy St. Arlington, VA 22217-5000	1

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-87-012	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* EXECUTION OF REAL-TIME PROTOTYPES		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR*(s)* LuQi		8. CONTRACT OR GRANT NUMBER*(s)*
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N: RR014-01 N0001487 WR4E011
11. CONTROLLING OFFICE NAME AND ADDRESS Chief of Naval Research Arlington, VA 22217		12. REPORT DATE April 1987 13. NUMBER OF PAGES 16
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

PSDL is a language for rapid prototyping, especially for systems with real-time constraints. This paper briefly introduces prototyping and PSDL, and describes how a PSDL prototype can be used to firm up the requirements of a software system. A scheme for making PSDL prototypes executable is outlined and the required support software is described.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Execution of Real-Time Prototypes

Luqi

Computer Science Department

Naval Postgraduate School

Monterey, CA 93943

ABSTRACT

PSDL is a language for rapid prototyping, especially for systems with real-time constraints. This paper briefly introduces prototyping and PSDL, and describes how a PSDL prototype can be used to firm up the requirements of a software system. A scheme for making PSDL prototypes executable is outlined and the required support software is described.

1. Introduction

Some important problems facing the computer software industry are achieving cost effective production of software systems and increasing the quality of software products with respect to meeting user requirements. Many software development methodologies have been proposed to approach this goal. Most of the well-known ones, such as Object Oriented Design (OOD) and the Jackson System Development Method (JSD) depend on the skill of individual designers at the level of manual work. These methods are labor intensive, and are too informal to guarantee any quality standards for the resulting design. These methods are unlikely to lead to any significant improvements in the reliability of software products.

Newly proposed software tools for software design, such as GENESIS [Ramamoorthy 85], SREM [Alford 77], the PAISLey operational approach [Zave 82], and DIANA [Evans 83] go one step further. Most of them are really software development environments consisting of many software tools for computer-aided software development. Some of them are designed to fit specific needs, eg. DIANA is a tool for the design of Ada systems. Even if we consider only the prototyping aspects of these tools, it is clear that complete automation of software development is still a distant goal. Most of these types of tools cannot be extended to the point of complete automation because of the lack of mathematical formalization in the related theoretical fields of software engineering.

In the rapid prototyping paradigm, the traditional software life cycle used in software design is replaced by a recently proposed alternative life cycle which consists of two phases: rapid prototyping and automatic program generation [Yeh 84]. Completely automatic generation of programs from very high level specifications is not currently practical. In our approach, program construction is speeded up by taking advantage of reusable software components drawn from a software base. The aspects of program construction that benefit from mechanical assistance are retrievals from the software base, generation of code for interconnecting available modules, and static task scheduling.

A rapid prototype is an executable model or a pilot version of the intended system. The construction activity leading to such a rapid prototype is called rapid prototyping [Neumann 82]. The use of rapid prototyping in requirements analysis

is described in Figure 1 [Luqi 87d].

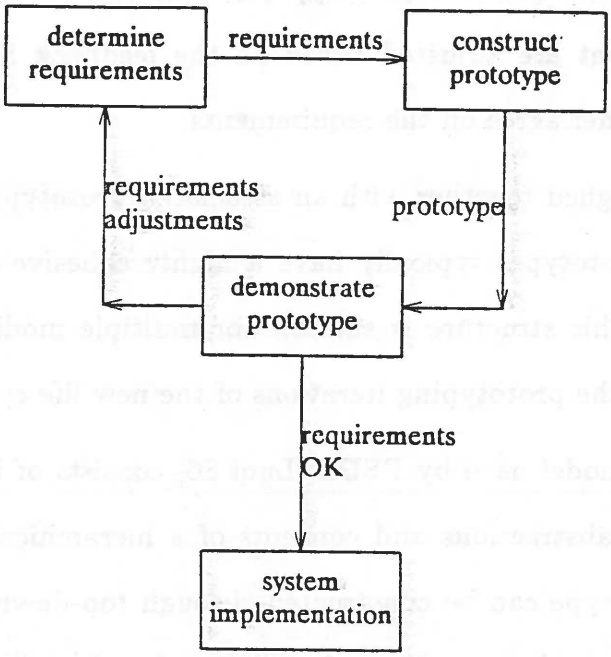

Figure 1 - Process of requirments determination and validation by prototyping

PSDL (Prototype System Description Language) has been designed for prototyping large software systems at the University of Minnesota. The language has special features particularly appropriate for real time system design.

PSDL prototypes can be used for requirements analysis and validation. A key issue in the design of large software systems is how to meet the requirements. Lack of agreement on the requirements as specified by user and analyzed by the designer usually increases the development costs and causes inconsistencies between the delivered system and user expectations [Yeh 83]. Because the user can usually recognize whether or not a working software system does what is needed, but usually can't describe the requirements accurately, prototypes are an effective means for achieving stable and accurate requirements. PSDL prototypes are especially well suited for this purpose because they are executable and are constructed at a specification level with requirements tracing. The prototype is used

in an iterative process of negotiation. The user describes the requirements, the designer interprets them and builds a prototype, and then asks user to criticize the result. The requirement are adjusted based on the feedback from the users until both users and designer agree on the requirements.

Since PSDL was designed together with an associated prototyping methodology [Luqi 87b], PSDL prototypes typically have a highly cohesive structure and few coupling problems. This structure is suitable for multiple modifications at a specification level during the prototyping iterations of the new life cycle.

The computational model used by PSDL [Luqi 86] consists of basic building blocks for describing the abstractions and concepts of a hierarchically structured prototype. A PSDL prototype can be constructed through top-down design. The PSDL non-procedural control constraints describe and enhance the real-time requirements at all layers of the hierarchical prototype.

The executable PSDL prototype is also used to check real-time requirements because the critical timing constraints and the most important concerns, eg. maximum execution time, minimum response time, and synchronization are very hard to validate without actually constructing a valid schedule and observing the execution of the prototype. Most real time systems are used to monitor and control physical processes external to the computer in the embedded system. The precision and accuracy requirements in the design of a real time control system complicate the demands on the execution of the designed software system. For these reasons, the design of real time systems imposes a different set of demands. The formal structure specifying the real time constraints in PSDL provides a basis for automating the production of code from the formal requirements specifications to the underlying programming language. The execution of PSDL prototypes helps to verify that the design of an embedded system with given timing constraints for

the components in the prototype will interact with its environment in a way that meets the timing constraints of the system as a whole. This is important because making a production quality implementation is very expensive, so that it is necessary to check that a design is feasible by using an inexpensive prototype before committing significant resources to an implementation.

PSDL is a language for describing prototypes of large software systems with real-time constraints [Luqi 86]. Such systems are modeled in PSDL as networks of operators communicating via data streams. The data streams can carry data values of an abstract data type [Guttag 78] as well as tokens representing exception conditions. Each type or operator is either composite or atomic. Composite operators are implemented by decomposing them into networks of more primitive operators using PSDL. Atomic operators are realized by retrieving an implementation from a software base [Yeh 84, Roussopoulos 85] containing reusable software components implemented in an underlying programming language. The reusable software components in the software base can be written in any general purpose programming language, provided that PSDL specifications for each module are included.

2. Prototype Execution

PSDL prototypes are executable if all required information is supplied [Luqi 87d] as indicated in Figure 2, CAPS (Computer Aided Prototyping System) Architecture, and the software base contains both normalized specifications and

implementations for all atomic operators and types [Luqi 87c].

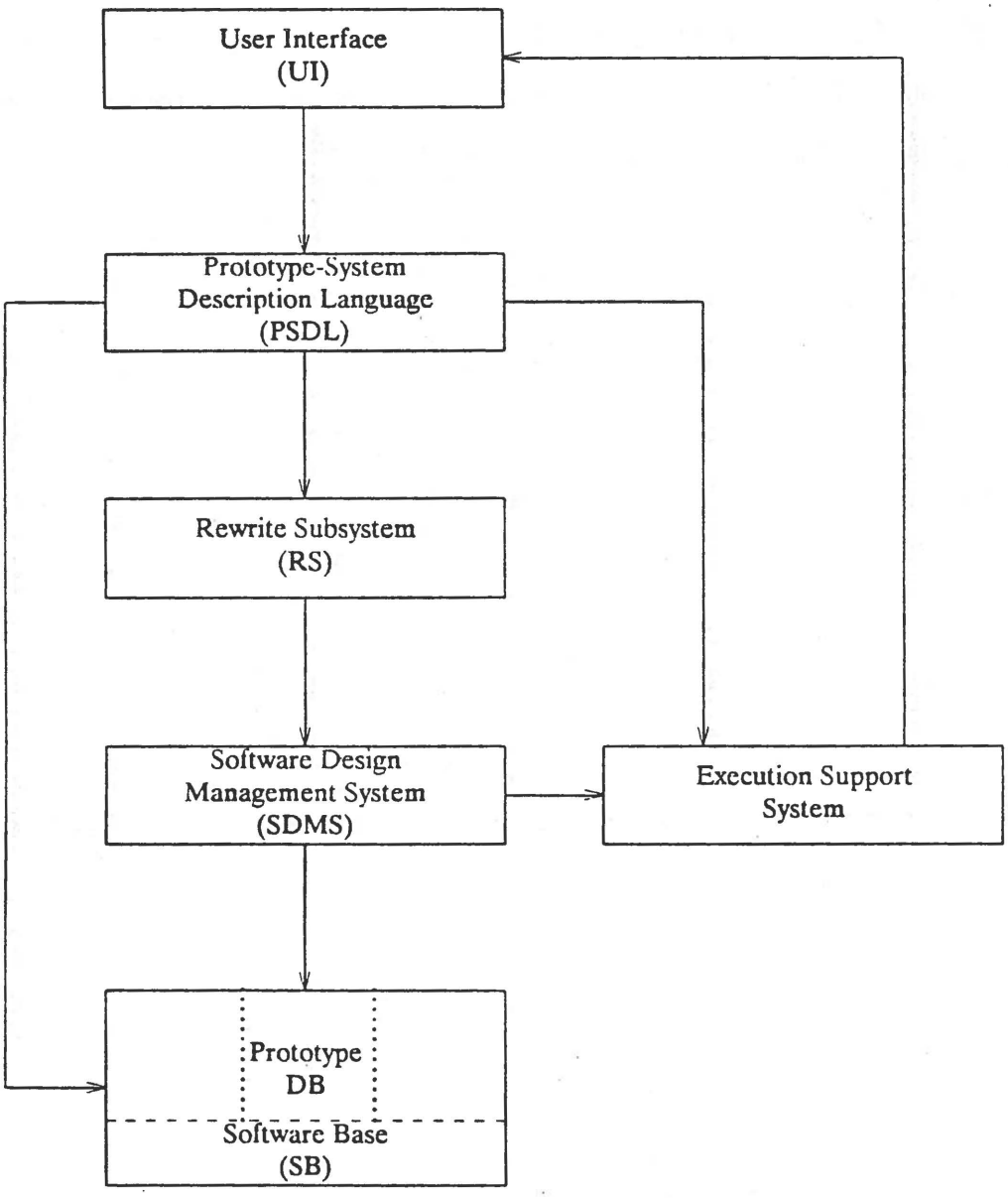

Figure 2- CAPS Architecture

To simplify the description of the PSDL translator we will assume here that Ada [Ada manual] is used for implementing both the reusable components in the software base and the PSDL execution support environment. The PSDL execution support system contains a static scheduler, a translator, and a dynamic scheduler. The static scheduler attempts to find a static schedule for the operators

with real time constraints. The translator augments the implementations of the atomic operators and types with code realizing the data streams and activation conditions, resulting in a program in the underlying programming language that can be compiled and executed. Execution is under the control of a dynamic scheduler, which also provides facilities for debugging and gathering statistics.

2.1. Static Scheduler

The static scheduler analyzes the real time constraints declared in the PSDL prototype, and attempts to find a static schedule meeting the timing constraints of the time critical operators. The operators that do not have real time constraints are handled by the dynamic scheduler, as explained in the next section. Time critical operators are either periodic or sporadic. Sporadic operators are implemented by their periodic equivalents.

The static scheduler partitions the set of periodic operators into non-overlapping HARMONIC BLOCKS, and constructs a STATIC SCHEDULE for each harmonic block. A harmonic block is a set of operators with the following properties:

(1) The periods of all of the operators in the set are exact multiples of the BASE PERIOD.

(2) One of the operators in the set has a period equal to the base period.

Harmonic blocks are not automatically disjoint (contrary to the assumption in [Mok 84a]). We have chosen to make them disjoint by placing each operator that satisfies the constraints for more than one harmonic block into the block with the longest possible base period, since we believe this heuristic will help to ease schedule congestion. A static schedule is a table giving the starting times and execution times for each operator in a harmonic block, and covers a length of time

equal to the least common multiple of all the periods in the block. The static schedule is constructed using algorithms similar to those of [Mok 84a]. The static scheduler uses a set of assumptions about the number and speed of the processors available and the cost of interprocessor communication, which must be provided by the designer.

Note that we treat each harmonic block as an independent scheduling problem. This is justified by the assumption that there will be at least one physical processor for each harmonic block. This assumption is reasonable because any schedule containing two operators with relatively prime periods is guaranteed to have periodic tight spots due to the beats between the two frequencies, leading to low utilization of the scheduled processor.

Sporadic operators are implemented as their periodic equivalents [Mok 84b]. The period P of such an equivalent periodic operator is given by the formula

$$P = \text{MIN}(\text{mcp}, \text{mrt} - \text{met})$$

where mcp is the MINIMUM CALLING PERIOD, mrt is the MAXIMUM RESPONSE TIME, and met is the MAXIMUM EXECUTION TIME. An equivalent periodic operator derived in this way has a deadline equal to the maximum execution time, which means that it must be scheduled to start at the beginning of each period (although a constant phase shift is allowable). A sporadic operator cannot meet its timing constraints unless P is greater than the maximum execution time of the operator.

2.2. Translator

The translator augments the code for the atomic operators to adapt them to the context in which they are going to operate. The augmentations have four main purposes: to implement PSDL data streams, to implement PSDL

conditionals, to implement PSDL timers, and to create exception closures.

Data streams are implemented using buffers. The buffers are at the receiving end of each data stream, and have a capacity of one data item each. The buffer for a sampled stream has a flag which is set whenever a new data value is received and is cleared whenever the data value is read. This flag is used in controlling data driven operators. The buffer for a data flow stream has a similar flag, which is set when the queue is full and cleared when it is empty. The difference between the two kinds of buffers is that an exception occurs if an attempt is made to write into a full data flow buffer or to read from an empty one, while reading or writing a sampled buffer will never cause an exception. The exceptions produced by the incorrect use of data flow buffers cause a transfer of control to the PSDL debugger.

PSDL conditionals are implemented using the conditionals of the underlying programming language. Triggering conditions are implemented by code of the following form:

IF triggering_condition THEN operator_body END

This is followed by an epilog which clears the input buffers and writes the outputs of the operator into the appropriate output buffers. If PSDL output guards are present, the output buffer operations are embedded in conditionals of the form

IF output_guard THEN output_buffer_operation END

The buffers are instances of a built-in data type which is implemented using a standard reusable module written in the underlying programming language. The multiple processor version of the PSDL translator uses a buffer type whose primitive operations are embedded in a monitor to provide mutual exclusion. The buffers are the only source of potential interference between processes, since all of the other data structures are local to a single process. The single processor

implementation does not need monitors for the buffers, because a simple implementation scheme using a single thread of control is most efficient for this case.

PSDL timers are implemented by a standard library package that communicates with a hardware clock. This package is included in any prototype that uses timers.

An exception closure is a package that handles all of the exceptions that may occur inside it. The PSDL translator produces exception closures by augmenting the code for atomic operators with handlers for any exceptions they may raise. These default handlers catch the exceptions and use the CREATE operation of the built-in EXCEPTION data type to create the required exception value, which is transmitted along the appropriate output data streams.

2.3. Dynamic Scheduler

The dynamic scheduler is a run-time executive with three main purposes: to schedule operators that are not time critical, to provide debugging facilities, and to gather statistics about the run-time characteristics of the prototype. In the case of a distributed implementation, there is an instance of the dynamic scheduler running on each processor.

PSDL assumes that time constraints are absolute if they are given. This requires the static scheduler to allocate processor time based on worst case execution times and firing frequencies. This policy results in plenty of spare processor time on the average, because worst case loads tend to be rare. The dynamic scheduler uses a simple strategy to utilize this spare capacity for operations that are not time critical.

During each base period the dynamic scheduler invokes the time critical operators in the order in which they are scheduled. When it runs out of things to

do, it checks to see if it has any time left, and if so it picks a non time-critical operator to execute. A simple round robin scheduling algorithm is used. Just before the end of the base period, the currently running operator is interrupted and the resumption point for the operator is saved. The interrupt is given sufficiently long before the end of the base period so that the currently running operator will have enough time to get out of any critical sections it may have entered. The only critical sections in the system are in the buffering primitives for reading values from data streams and writing values into data streams. These critical sections are short, and have fixed upper bounds on their execution times.

The debugging facilities are fairly conventional. Breakpoints can be attached to operators, and can be conditional with respect to a PSDL predicate. Selected inputs or outputs of an operator can be traced, resulting in a display of the values and their associated arrival or departure times. Commands for inserting and deleting values in data streams are provided. The facilities for gathering statistics include commands for monitoring both frequencies and timing information. Frequency statistics include the number of values that pass down a data stream, the number of times an operation fires, the number of times an exception occurs, etc. Timing statistics include minimum, average, standard deviation, and maximum times for the execution, response, or interval between firing of an operator. These statistics are intended primarily for feasibility and performance studies.

3. Conclusions

There are many aspects of software requirements which can be most effectively validated by user inspection of a running prototype, such as the appropriateness of a given user interface, or the correct description of an existing hardware interface. Executing prototypes of the novel or difficult parts of a com-

plicated system can significantly increase the confidence that the system can in fact be built, before significant resources have been committed to the development effort. Cost estimates can be improved by using a prototype, since the cost of designing the intended system is usually proportional to the cost of the rapid prototype. Performance bottlenecks can be found during the execution of the prototype by collecting statistics on module execution frequencies.

Our initial investigation leads us to conclude that an execution support system for PSDL is feasible, and that such a software tool is currently the most practical way to support rapid prototyping for real-time systems. This together with the features of PSDL for large scale software design makes PSDL a good candidate for inclusion in an advanced Ada programming environment. At the current point in time, we have a conceptual design for the PSDL execution support system, and the implementation of the PSDL translator is under way.

The PSDL language, its associated methodology, and programming environment apply well to the design of Ada software systems. The demand for large scale Ada software systems is increasing dramatically. Real time systems have particularly strict requirements on accuracy and precision. A rapid prototyping environment for creating and modifying an executable prototype is needed. The design of PSDL, its prototyping methodology, and the use of reusable components from a software base make highly automated software tools practical. An experienced PSDL user should be able to rapidly construct a prototype significantly faster than an experienced Ada-user.

The use of PSDL for prototype construction should be much easier and simpler than the direct use of Ada. PSDL has selected and transformed all the good language features of Ada primitive constructs into a small and a simple set of PSDL language constructs which is convenient for the designer. It is simpler to

describe the structure of a system and the relation between system components in PSDL than in Ada since PSDL allows a designer express his thoughts at a specification or a design level. The abstractions of PSDL are tailored to describing real-time systems, and allow the designer to express his thoughts clearly and quickly by eliminating many lower level details from his consideration. The computational model of PSDL forces all interactions between models to be explicit. All state variables are local to some component, thus confining the effects of state changes. This helps designer understanding by eliminating hidden interactions on the large scale, while allowing the efficiencies of imperative programming inside individual components. The important points are that the software tools and the prototyping methodology of PSDL lead to a well structured prototype and that the resulting PSDL prototype is executable. PSDL components can be mapped into Ada directly. Ada is a large and powerful programming language. It is a good underlying programming or an implementation language for PSDL. However, it is too hard and too cumbersome to use as a design language. The mapping between PSDL and Ada and the use of the reusable Ada components are the keys to making PSDL prototypes executable and useful in large Ada projects.

4. References

[Ada manual]

Ada Programming Language, DoD, ANSI/MIL-STD-1815A, 1983.

[Alford 77]

A. W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements", IEEE Transactions on Software Engineering, vol. SE-3, no. 1, 60-68, Jan 1977.

[Evans 83]

A. Evans, K. J. Butler, G. Goos, and W. A. Wulf, "DIANA Reference Manual", Revision 3,

Tartan Laboratories Inc., Pittsburgh, Penn., 1983.

[Guttag 78]

J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract Data Types and Software Validation", CACM 21, 12, Dec. 1978.

[Luqi 86]

Luqi "Rapid Prototyping for Large Software Systems", Ph. D. Thesis, University of Minnesota, May 1986.

[Luqi 87a]

Luqi, V. Berzins, R. Yeh, "A Prototyping Language for Real-Time Software", to appear in IEEE TSE, 1987.

[Luqi 87b]

Luqi, V. Berzins, "Rapid Prototyping of Real-Time Software", revised for IEEE SOFTWARE, 1987.

[Luqi 87c]

Luqi "Normalized Specifications for Identifying Reusable Software", submitted to ACM-IEEE Computer Society 1987 Fall Joint Computer Conference, October 1987.

[Luqi 87d]

Luqi, M. Ketabchi, "A Computer Aided Prototyping System", submitted to First International Workshop on Computer-Aided Software Engineering, May, 1987.

[Mok 84a]

A. K. Mok, "The Design of Real-Time Programming Systems Based on Process Models", IEEE, 1984.

[Mok 84b]

A. K. Mok, "The Decomposition of Real-Time System Requirements into Process Models", IEEE Proc. of the 1984 Real Time Systems Symposium, Dec. 1984, pp. 125-133.

[Neumann 82]

P. G. Neumann, ed., "Special Issue on Rapid Prototyping", SIGSOFT, Dec 82.

[Ramamoorthy 85]

C. V. Ramamoorthy, Y. Usuda, W. Tsai, and A. Prakash, "GENESIS: An Integrated Environment for Supporting Development and Evolution of Software", PROC. COMPSAC 85.

[Roussopoulos 85]

N. Roussopoulos, "Architectural Design of the SBMS", Quarterly Report for the STARS SB/SBMS Project, Dept. of Computer Science, University of Maryland, April 1985.

[Yeh 83]

R. T. Yeh, "Software Engineering", IEEE Spectrum, Nov. 1983, p. 91-94.

[Yeh 84]

R. T. Yeh, R. Mittermeir, N. Roussopoulos, and J. Reed, "A Programming Environment Framework Based on Reusability", Proc. Int. Conf. on Data Engineering, Apr. 1984.

[Zave 82]

P. Zave, "An Operational Approach to Requirements Specifications for Embedded Systems", IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, 250-269, 1982.

Initial Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analyses 2000 N. Beauregard Street Alexandria, VA 22311	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
LuQi Code 52Lq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	100
Office of Naval Research 800 N. Quincy St. Arlington, VA 22217-5000	1